NTATION PAGE

Form Approved
OMB No. 0704-0188

AD-A217 765

ORT DATE	3. REPORT TYPE AND DATES COVERED
January 15, 1990	FINAL Report, 1 Mar 87 thru 31 Oct 89

4. TITLE AND SUBTITLE
RESEARCH IN PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

5. FUNDING NUMBERS
AFOSR-87-0130
61102F 2304/A2

6. AUTHOR(S)
Dr. Victor R. Basili
John D. Gannon
Marvin V. Zelkowitz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
Department of Computer Sccience
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-87-0130

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research
Building 410
Bolling AFB, DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFOSR-TR- 90-0077

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report summarizes the activities from this grant during the period March 1, 1987 through October 31, 1989. During this period, we have pursued research on software engineering environments, specification and implementation languages and compiler technology. The projects presented in this report encompass developing and integrating the concepts and models used in the TAME measurement environment, using syntax-editing technology to develop formal specifications, investigating the impact of functional specification and development on software construction, designing and evaluating a new exception handling mechanism, and transforming computations for single processors to execute efficiently on non-shared memory multiprocessors.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

FINAL TECHNICAL SUMMARY

Research in Programming Languages and Software Engineering

AFOSR Grant 87-0130
March 1, 1987 – October 31, 1989

Principal Investigators

Victor R. Basili
John D. Gannon
Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Date: January 15, 1990

Abstract

This report summarizes the activities from this grant during the period March 1, 1987 through October 31, 1989. During this period, we have pursued research on software engineering environments, specification and implementation languages, and compiler technology. The projects presented in this report encompass developing and integrating the concepts and models used in the TAME measurement environment, using syntax-editing technology to develop formal specifications, investigating the impact of functional specification and development on software construction, designing and evaluating a new exception handling mechanism, and transforming computations for single processors to execute efficiently on non-shared memory multiprocessors.

This report represents the technical summary of AFOSR grant 87-0130 to the Department of Computer Science of the University of Maryland for the period July 1, 1987 until October 31, 1989. Copies of all relevant papers have previously been forwarded to AFOSR as they appeared. The activities from March 1, 1987 through June 30, 1988 have already been reported to AFOSR, and a copy of that report is attached as an appendix to this report. Since this is a multi-investigator effort, some of these summaries represent the completion of previous projects while others represent ideas just now being developed.

This research was under the direction of the principal investigators Dr. Victor Basili, Dr. John Gannon and Dr. Marvin Zelkowitz.

1. The TAME Environment

During this past year, the emphasis on the TAME project has been on formalization and integration of its concepts and models. These models are stored in an Experience Base which is part of an Experience Factory [Basili89]. The Experience Factory allows previously defined models to be reused, after some possible modification, in future projects.

More specifically, logical and physical representations for process, product and quality models have been studied and the Goal/Question/Metric (GQM) paradigm has been packaged to allow the integration with these various models. Thus the primary parameters of a goal, its purpose and perspective, can be written as an ordered pair: a model of the object of study and a model of the quality perspectives of interest. The secondary parameters, the focus and the point of view, are defined as modifiers of the object and quality perspective, respectively. Thus a particular GQM model ties together a set of models so that it can prescribe the data that needs to be collected, organize its collection and allow for the interpretation of the data within the context of the goal.

1.1. Goals

The current version of the goal template is defined as follows:

Purpose: Analyze (objects: a single object, a set of objects, the relationship between a set of objects,...) for the purpose(s) of (focus: characterization, evaluation, prediction, motivation, comparison, improvement,...)

Perspective: with respect to a model of the (cost, effectiveness, correctness, defects, changes, product metrics, reliability, etc.) from the point of view of the (developer, manager, customer, corporation, researcher, etc)

Environment: in the following context (process models, people factors, application types, methods, tools, constraints, etc.)

For example, we might decide to analyze the system test method for the purposes of evaluation and improvement with respect to a model of defects that persist from one lifecycle stage to another from the point of view of the developer in the NASA/GSFC environment (i.e., a version of the waterfall model applied to ground support software for satellites running on a DEC 780 under VMS, etc.).

1.2. Questions

Guidelines for posing questions associated with the goals have been standardized. While questions related to products, processes, and resource models necessarily differ, our guidelines suggest a set of common subgoals that need to be addressed: quantitative definition of a product, process, or model; identification of quality perspectives; and specification of feedback mechanisms related to the quality perspectives.

Guidelines for defining products, processes, or models

Products are characterized quantitatively by the following components:

physical attributes (size, complexity, etc.),
cost (effort, computer time, etc.),
changes and defects (errors, faults, failures, adaptations, and enhancements), and
context (an operational profile of the user community).

Process models are characterized quantitatively by:

process conformance (how well the process is performed) and
domain conformance (an assessment of the object to which the process is applied and the process performer's knowledge of the object).

Models are characterized quantitatively by:

model complexity (the number of parameters, ranges on those parameters, etc.),
model appropriateness (an analysis of how relevant the model is for that object),
errors made in applying the model (if relevant),
model conformance (an assessment of how well the model is adhered to), and
cost (of applying the model in terms of effort, computer time, etc.).

Guidelines for identifying quality perspectives

Quality perspectives may be quantitative or qualitative (e.g., reliability or user friendliness). They provide an interpretation of the model relative to the data collected and help identify metrics. Quality perspectives include questions related to

the models used (a quantitative specification of the quality perspective),
the appropriateness of the model for the particular project environment,
the validity of the data collected,
the model effectiveness (a quantitative characterization of the quality of the results produced according to this model), and
(optionally) a substantiation of the model (an alternative model to help evaluate the results of the primary model).

Guidelines for specifying feedback mechanisms

Feedback information includes questions related to improving the products, processes, various models and GQMs based upon the quality perspective. It should also include lessons learned as well as information that will change the models. Feedback very often references other factors not explicitly mentioned in the definition of the product or quality perspective. In these cases it should be checked that these factors exist either in the environment section (when there is an attempt to evaluate against a data base) or in the definition of the product section (when there is a need to examine the model of the project).

1.3. Sample Quality Perspective Model

The sample goal above sought to analyze the system test process for the purposes of evaluation and improvement with respect to a model of defects that persist from one lifecycle stage to another from the point of view of the developer. After developing a model of the developer's system test process and evaluating the process and domain conformance for the current project, we need to define a quality perspective for faults found in various lifecycle stages, We measure the faults found in the system under development:

Es = #faults found in system test in this project
Ea = #faults found in acceptance test in this project
Eo = #faults found in operation in this project

and compare them to the faults discovered in a set of previous projects {Pi}

PEs = average #faults found in system test in {Pi}
PEa = average #faults found in acceptance test in {Pi}
PEo = average #faults found in operation in {Pi}

Simple Model of Defect Slippage

The ratio of faults present during system test on this project to faults found from system test through operation.

$$REs = Es/(Es+Ea+Eo)$$

The ratio of faults present during system test in the standard projects to faults found starting at system test.

$$RPEs = PEs/(PEs+PEa+PEo)$$

The relationship of system test performance on this project compared to the standard projects.

$$QEs = REs/RPEs$$

Simple Feedback Interpretation

if QEs $<$ 1 then method good
elseif QEs $=$ 1 then
    method average
    if cost lower than normal then method cost effective
elseif QEs $>$ 1 then
    if process conformance and domain conformance good then method poor
    else method different

Clearly this process should be done not only at this high level but for each class of fault, the total cost to isolate and fix a fault in total and by class of fault. We can do the calculation by error and failure category as well. We can also look at normalized defects by size.

1.4. Current Activities

We are in the process of generalizing this simple model into a class of models that can be parameterized to include other phases as well as various classes of defects. This involves an understanding of how the questions and data collected change as the parameters change, how the model may be represented so as to be instantiated properly using the different parameters, and how the interpretations might change appropriately.

To establish the models as part of the Experience Base, we are working on a top level architecture that represents and integrates the various models. Mechanisms under study include the use of an object oriented language for representing the quality models, a process model language for representing processes, and the use of expert system shells for representing and interpreting the data, and integrating the various models.

2. Integrated Environments

2.1. Syntax–Based Program Editing

The SUPPORT environment was developed to study semantic interactions in the development of source programs. SUPPORT uses a syntax–based (i.e, language–based) editing paradigm for program generation. Pure syntax–based editing is a simple macro–like substitution and such macro substitutions exist in several conventional editors. For example, *Emacs* and Digital's *LBE* (Language Based Editor) both permit such substitutions anywhere in a program. However, SUPPORT's features go far beyond simple substitution: screen layout (e.g. unparsing the program tree to a "pretty–printed" display), semantic checking (e.g., type consistency between declarations and uses of variables), testing (e.g., coverage metrics and test oracles), and source–level debugging. SUPPORT offers an integrated package or environment of editor, interpreter, and debugging and testing tools [Zelkowitz89a, Zelkowitz89b]. The proposed advantages for such an editor were many:

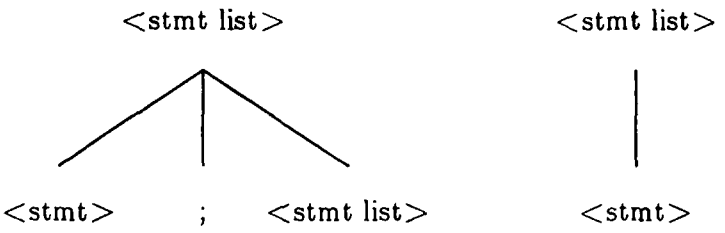
(1) Productivity would increase because source program generation would be efficient (a single mouse or function key click would generate an entire construct) and numerous errors (e.g., missing paired symbols and type–inconsistent expressions) either could not occur or would immediately be detected.

(2) Predefined screen layout would provide a uniform program structure, enhancing a programmer's comprehension and freeing him from maintaining a format manually.

(3) The integrated package of tools enables testing and debugging to proceed in the same notation as program development.

Our experiences with SUPPORT showed many of these advantages to be illusory [Zelkowitz90a]. While most program construction activities are performed more quickly using syntax–based editors, several common constructs present problems that reduce the gains. For example, in adding an **if** statement, the editor chooses between implicitly inserting an **else** clause and (possibly) having a programmer delete it, and requiring a programmer to add necessary **else** clauses explicitly. In either case, the editor is wrong about half of the time. A more serious problem is that the editor expects syntactic units to be added in the manner in which a top–down parser would encounter them; however, programmers usually write source

code from left to right. For adding new statements, there is not much difference between sequential insertion and a top–down parse:



In either case, statements are processed left to right. However, inserting expressions such as A+B*C requires that symbols be selected in prefix order (e.g., "+A*BC"). The need for an internal parser (such as the SUPPORT's LALR parser) to allow infix notation to be automatically parsed into the appropriate program tree structures is crucial to the success of all such editors.

Error prevention and early detection do not benefit experienced programmers as much as novices. Experienced programmers generally do not make many syntax errors. When they do make errors, experienced programmers may prefer to finish entering their code before repairing an error. However, with a syntax–based editor, only correct syntax can be entered. The system will usually halt and beep until corrective action is taken. Thus there is a disruption in a train of thought where some semantic issue needs to be put aside (and forgotten?) in order to fix some simple syntax.

Although it generates nicely indented displays of source code, predefined screen layout cannot handle special cases (e.g., a sequence of conditional statements) or text not defined by the language's BNF (e.g., comments).

While source–level testing and debugging is very convenient, they are not unique to syntax–based editors. One needs an integrated framework and data repository for a source program. The current interest in CASE (Computer Aided Software Engineering) tools exemplifies this, and SUPPORT is simply a CASE tool with a syntax–based editor for a base.

Our experiences with SUPPORT are by no means unique. For example, Mentor, initially developed at INRIA, has had a similar pattern of development and use:

> Novices used menus but experienced programmers rarely did;
>
> Experienced programmers wanted the full–screen Emacs editor for textual input and modification (providing functionality similar to SUPPORT's editor) using automatic parsing and unparsing of the Mentor input; and
>
> Switching between Mentor and Emacs was difficult due to the inherent problems in placement of comments.

On the other hand, Mentor was a powerful source code maintenance system due to the integration of many program analysis tools which shared semantic information about a program. But just as in SUPPORT's case, such tools are mostly a function of Mentor being an integrated environment and not simply an editor.

In conclusion, the drawbacks seem to be as serious as the advantages in syntax–based editing, which probably explains their lack of growth and popularity since the early 1980's. Since source code development is often estimated as being 15% of total life cycle costs, even if an editor reduced coding time to zero, productivity improvements would remain small.

2.2. Creating Specifications

The previous discussion indicates that while syntax–based editing of source programs is a powerful technique, it has minimal effect upon productivity. However, since requirements, specification, and coding consume 75% of development costs, improving those phases of the life cycle might have more impact on productivity. In addition, improving the correspondence between specifications and design and between design and source code would eliminate interface errors, hence decreasing the effort needed in testing and further increasing productivity.

We are investigating syntax–based editing of specifications. By modifying the grammar of the language processed by SUPPORT, SUPPORT became an interface "shell" for a series of integrated environments (e.g., AS*). Much of software design consists of the creation of complex data objects, usually referred to as abstract data types, and the definition of functions that operate on these abstract objects. Using algebraic specification technology, a series of equations relates the operations of the abstract type to each other. These equations, in the proper form, can be viewed as a term rewriting systems. The use of the Knuth Bendix algorithm defines a proof of adequacy of the resulting algebraic equations by showing the equivalence of supposedly equal terms to the same ground (e.g., constant) terms. However, since the Knuth Bendix algorithm is based upon an ordering transformation from one term to a "simpler" term, the algorithm defines an operation that can be "executed" and proven to terminate. Therefore, any set of axioms that is "Knuth Bendix" can be translated into a series of transformations that can be executed in some programming language. In addition, the executable program provides the basis of a test "oracle" for judging the correctness of enhancements.

Our model of a specification follows closely with the initial algebra approach of other term rewriting systems. An AS* specification contains 3 features: a set of *sort* names which define new abstract objects and their *constructors;* a *signature* which defines a set of *defined operations* and constructors for manipulating the abstract objects; and a set of oriented *equations* (or *axioms*) which relate the defined operations and constructors to each other, and Details of AS* are explained in [Antoy90].

A prototype implementation of the AS* system has been constructed and executes on the SUN 3 workstation under Berkeley UNIX 4.3. The four components are described below.

(1) **AS/SUPPORT** provides syntax–based editing capabilities for creating specifications. AS/SUPPORT guarantees that all sort definitions have syntactic consistency with the underlying sort syntax. After the user builds a sort, AS/SUPPORT translates his specification into a format suitable for Prolog and invokes AS/VERIFY as a subprocess.

(2) **AS/VERIFY** checks that the specification is executable. Inadequate axioms are highlighted to allow the user to change the specifications interactively. The Knuth–Bendix algorithm either shows convergence of the axioms or indicates additional axioms which are needed; however, it may not indicate when sufficient axioms have been added in the case of not converging rapidly enough (the usual problem with undecidability results). Other verifiers usually interact with a user who manually indicates approval to continue the process or terminate.

(3) **AS/PC** is a translator that converts specifications into standard Pascal source programs.

(4) **PC** is the standard system Pascal compiler. At this point, the specifications have been converted to standard Pascal, and any comparable compiler can be used for compilation and execution.

This project is part of an ongoing effort in understanding formal specifications. Current efforts are related to developing axioms that can be converted to executable code most efficiently. We are also in the process of using AS* in enhancing existing Pascal programs with new features in order to test this methodology in a realistic setting.

AS* addresses many of the important specifications and code reuse problems today. Specifications are formal, yet executable and can easily be mapped into a variety of programming languages. Maintenance is enhanced via tests on the existing source program. Determining properties of specific abstract data type implementations is explicit and should help in the necessary process of understanding and reusing source program libraries.

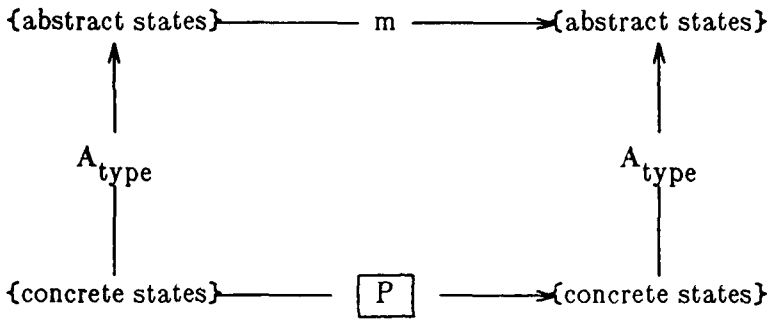
3. Program Verification

During the past fifteen years, Mills and others have been developing a theory of functional program semantics. A program is specified as a function or relation that defines a correspondence from one domain to another. A proof methodology was developed for formally showing that a given program is consistent with its specifications. As part of our effort, we have developed mathematical foundations for stepwise refinement and data abstraction [Mills89] [Zelkowitz90b].

3.1. Functional Properties

Understanding a program as a mathematical object is understanding the functional behavior it induces in a computer. An execution state is a relation or function whose domain is the identifiers of a program and whose range is the values attached to those identifiers. The meaning of a program is a mathematical relation or function, a set of ordered pairs that defines a correspondence between one state (the inputs) and another state (the outputs). Program P is correct with respect to specification relation r if, for every $x \in$ domain(r), P produces some member of the range of r which corresponds to x. The problems we try to solve are: given a program, find and verify its meaning; and given a meaning, derive a program with that meaning. The entire mathematical basis for our study rests on just five discrete mathematical structures of character data: strings, lists, sets, relations, and functions. These five structures are not only sufficient to deal with program correctness and program design, but also admit treatment at various levels of formality with a mixture of English and mathematical notation.

Following the work of Hoare and the designers of Alphard, we have developed a proof theory for data abstractions using functional semantics. The essence of data abstraction is captured by a diagram showing the relationship between the *concrete* objects manipulated by procedures (e.g., P), and the *abstract* objects the programmer manipulates with abstract operations (e.g., m) to achieve a solution. A *representation mapping*, denoted A_{type}, is defined between the values of concrete objects and the values of the corresponding abstract objects. By convention, for objects common to the concrete and abstract worlds, the representation mapping is identity. Then the representation mapping can be extended to map any concrete state to an abstract state.



Intuitively, an implementation is correct if its data objects are manipulated in such a way that the abstract objects to which they correspond, appear to be transformed according to the abstract operations. To decide if this property holds, we show that the diagram commutes.

$$A_{type} \; o \; m \subseteq [P] \; o \; A_{type}$$

Of course abstract operations like m do not really exist except in users' minds. Procedures implementing abstract operations are written with two sets of comments, abstract and concrete. The abstract comments are for users so they need not examine the code (or even the concrete comments that document it). The abstract comments replace the abstract operations in demonstrations that diagrams commute. If the implementation has been done properly, the abstract comment can be believed, and used in proofs at the abstract level.

3.2. Beyond Functional Properties

We have begun to extend the functional paradigm into the realm of requirements analysis. By extending the definition of program correctness, we can develop a framework for discussing life cycle models and then develop an evaluation procedure that enables us to compare different program solutions for a given specification.

Generally, there are several (actually many) feasible solutions to a given specification, any one of which might satisfy our functional constraints. We usually under specify a set of requirements and any program that meets those minimal requirements is deemed acceptable. In addition, large systems have other requirements beyond their functionality: cost to build, size and speed performance constraints, reliability criteria, etc. Therefore, given a basic specification, there generally exists a set of specifications, each being different from the others and each consistent with the basic specification. Each one could possibly lead to different correct implementations. We would like to be able to understand this set of specifications and to be able to compare them in order to determine which one from the set best meets the user's needs.

We now summarize our extended model.

Def: Let *PU* be the *problem universe* of specification functions. That is, if f is a specification function, $f \in PU$.

Def: Let *SU* be the *solution universe*. *SU* represents algorithm designs. For example, if $f \in PU$ is to sort an array, then *bubble-sort*, *selection-sort* and *quick-sort* would all be members of

SU.

Def: *Scaling function:* Assume there is a function S such that if $a \in PU$, then $S(a) \in [0..1]$. Scaling function S determines how good a specification we have. If $S(a)=0$, then we have a "useless" specification and if $S(a)=1$ then our specification is optimal and cannot be improved.

Def: *Solves:* Let A and B be members of PU, and let S be a scaling function. We say that A *solves*$_S$ B if and only if $S(A) \geq S(B)$, i.e., A is an improved specification to B.

Def: *Specifies:* Let $p \in PU$ and $P \in SU$ be the abstract function and concrete solution and let r be the data representation function. $r \circ p \subseteq [P] \circ r$. Define the exact specification P' for P as that function such that $r \circ P' = [P] \circ r$.

With our concept of exact specifications, we can now extend the previous definition of functional correctness:

Def: *Correctness:* Solution X is *correct* with respect to specification function R and scaling function S if and only if X' *solves*$_S$ R.

It is important to show that we have not deviated from the previous definitions in the functional correctness methodology. Therefore, we need to show that our definition of correctness is consistent with previous formulations. We give this as the following theorem:

Theorem: Given specification function R, and solution P then $R \subseteq [P]$ if and only if for an appropriate scaling function S, P' *solves*$_S$ R. By defining the scaling function as $S(X)=1$ *if* $R \subseteq [X]$ *else* 0, the proof follows.

As stated previously, functionality is insufficient as the only specification criterion. There are at least three other classes of attributes: *performance* (e.g., size, execution speed, disk usage), *reliability* (e.g., fault tolerance, accuracy, safety) and *development* (e.g., cost to build, time, personnel costs).

We extend the definition of a specification function to include a vector B of *basic specifications*, each B_i is an *attribute* of the specification. Thus PU is a set of vectors. Similarly, we extend S to be a vector of *scaling functions*. If $x \in PU$ and $y \in PU$ then x *solves*$_S$ y if and only if, for all i, $S_i(x_i) \geq S_i(y_i)$.

Given a specification vector f, a scaling vector S and feasible programs P and Q such that both $r \circ f \subseteq [P] \circ r$ and $r \circ f \subseteq [Q] \circ r$, we would like to determine which solution is preferable. Obviously, if P' *solves* Q', or if Q' *solves* P', then our choice would be obvious. However, such choices rarely occur in practice. All too typically, one solution might excel on some attribute (e.g., execution speed) while the other might excel on another (e.g., low cost to build). Comparing the relative importance requires a further extension to this model.

Def: *Constraint set:* If we have n attributes, let S be a vector of scaling functions. Consider a third vector of weights W such that each $w_i \in [0..1]$ and $\Sigma w_i = 1$. We will call $<S, W>$ the *constraint set* for a specification.

Def: *Performance level:* Given a basic requirement B and constraints $<S, W>$ where S is a vector of scaling functions and W is a vector of weights, define the *performance level* PL of B

relative to $<S,W>$ as $PL(B,S,W) = \Sigma\ (w_i * S_i(B_i))$.

Given a basic specification B and a constraint set $<S,W>$, we can now discuss the relative merits of alternative solutions. If x and y are both to be feasible designs, then, at the least, they must both satisfy the specifications i.e., we must have both x' $solves_S$ B and y' $solves_S$ B. In addition we would like to use the solution with the greater performance level. We call this the ***improves relation.***

Def: *Improves:* Given a basic specification $B \in PU$, *constraint set* $<S,W>$ and designs x and y such that $x,y \in SU$, we state that x' *improves* y' with respect to $<B,S,W>$ if and only if

(1) x' $solves_S$ B and y' $solves_S$ B and

(2) $PL(x', S, W) > PL(y', S, W)$.

Note that we are using a very simple sum of weights measure in order to compute the performance level. We recognize the simplicity of this approach; however, even with such a simple model we achieve interesting results. We plan to investigate other definitions later. For example, we can plot each of the attribute values on a circular graph and then look at the area covered by each potential solution (e.g., similar to Kiviat graphs for system performance evaluation). This and other approaches will be studied.

What is important to realize, however, is that our definition of *improves* depends only upon a definition of *performance level* to compare two solutions, not on the details of how the two vectors are compared. What follows will remain true regardless of the underlying metric used in the comparison.

Using this model we can apply it in diverse applications. Two such applications are described in [Cardenas90]:

(1) If $P(S)$ is the projection of vector S onto a subset of attributes, we can define a prototype as:

Def: *Prototype:* Let $B \in PU$ be a basic specification and let $<S,W>$ be a constraint set. $X \in SU$ is a *prototype* of B if and only if there is some binary vector v such that $P_v(X')$ $solves_{P(S)}$ $P_v(B)$.

Given this definition, we now have a formal model for life cycle processes, such as the waterfall model or the spiral model.

(2) Given several alternative designs to solve a problem, we can use the performance level measure as a quality metric. For example, we propose a sorting problem, and then give five applications and five alternative algorithms and show that each application has a different best solution based upon criteria other than functionality. In a second similar example, we propose a hardware and software solution to a disk file algorithm and choose among four competing designs.

We have looked at an aspect of functional specifications and developed an evaluation criteria for comparing solutions to a given set of requirements. Using this model we can also develop a framework for classifying and describing various life cycle models. While the work is still preliminary, we believe that we have an important model that can easily be expanded. Various definitions of *performance level* need to be studied in order to best approximate true system design. In addition, as shown by the various examples, the model is applicable in

various application domains. Other such examples need to be developed.

4. Data–Oriented Exception Handling in Ada

Exception handling mechanisms were added to programming languages to segregate normal algorithmic processing from error processing. However, there is little agreement about what events constitute "exceptional conditions." Furthermore, binding exceptions to handlers by attaching the latter to a program's control components clutters source text in much the same way that testing operations' input and result parameters does.

Ada is the first widely–used language since PL/I to include exception handling features, and its design reflects many improvements. However, Ada's exception handling mechanism might still be altered to improve our ability to understand and validate programs. Although handlers appear at the end of any block after the block's algorithmic code, introducing blocks in the middle of statement lists to attach handlers inserts error handling code in the middle of algorithmic code. Handlers and packages are not well integrated since such handlers apply only to the initialization sequences of the packages and not the operations in the package bodies. If a subprogram terminates because an exception is raised, then the values of out and in–out aggregate parameters differ since a compiler may adopt either reference or copy–in, copy–out implementation strategies. Automatic propagation of unhandled exceptions (rather than explicit propagation of re–named exceptions) makes it impossible to determine statically the set of handlers that can catch an exception.

We have redesigned Ada's exception handling features and implemented an preprocessor to translate our features into Ada [Cui89, Cui90]. Like Black, we raise exceptions only in response to implementation insufficiencies, situations in which the storage reserved for an object is inadequate to represent its value or when an operation's performance constraints cannot be met. Our view of exceptional conditions as implementation insufficiencies results in a mechanism that is tightly coupled with Ada's package construct. Exceptions are defined and raised only in packages because such conditions are defined in terms of an object's representation, which can be manipulated only in a package body. Each data object declared has its own set of (exception, handler) binding pairs specified in its declaration.

4.1. Empirical Studies

We analyzed about two dozen programs in the Simtel20 Ada Repository and categorized their uses of exceptions. Most handlers take simple actions, such as propagating the raised exception upward until termination, or just printing error messages. Of the 26 programs analyzed, 15 have trivial exception handling (2 have no exception handling and another 13 just print messages and propagate errors). Only 2 programs have more than 15% of their exception handling statements that implement algorithms to recover from errors. Few programs have deeply nested exception handlers, and not surprisingly, these programs are very hard to understand.

To determine the effects of transforming a procedure with control–oriented exception handling into one with data–oriented exception handing, we examined one of the two programs with non–trivial handlers (This program calculates the correct compilation order of Ada source program units.) and rewrote its implementation. As expected, the original procedure and the revised version (including the extra procedures) have about the same number of statements. However, the new version breaks the original code into three smaller procedures, resulting in better modularity and functionality. As for the complexity, the original version has up to three levels of nested handlers, one of which was unreachable. In contrast, the

revised version has no handler code mixed with the main code of computation, thus emphasizing the main algorithm and enhancing readability. Sample executions on worst–case data show no difference in execution time between the two versions and approximately a 5% space penalty in the compiled code of the data–oriented version.

We conducted two other studies to investigate the effects of different exception handling mechanisms on program construction, comprehension, and modification. The subjects were were experienced programmers working for commercial software companies. These studies were performed on relatively small programs to help substantiate claims about benefits provided by data–oriented exception handling. Although the results cannot be generalized to large systems, the data encourages us to apply our methods to these systems.

In our first study, we tested four pairs of null and alternative hypotheses that compare the sizes of programs, numbers of statements per subprogram, and maximum and average nesting depths of statements. We conjectured that programs with data–oriented exception handling would have smaller measures of each of these metrics than similar programs with control–oriented exception handling. Subjects solved the same problem twice, first with Ada and then with our version of Ada with data–oriented exception handling. Although we evaluate the results of this study as if it were a controlled experiment, substantial learning effects may bias the results. Using data–oriented exception handling results in simpler (i.e., less deeply nested) and smaller code.

Our second study was designed to test how the choice of the different exception handling mechanisms would affect program comprehension and modification. Subjects read a two 4–5 page programs and answered several questions. In order to investigate the effect of different exception handling mechanisms on program comprehension and modification, we designed two equivalent versions of programs for each of the problems: a control–oriented exception handling version (C) and a data–oriented exception handling version (D), respectively. A subject assigned to work on version C of one problem studied version D of the other problem, and vice versa. We tested two pairs of null and alternative hypotheses that compared test scores and solution times. We conjectured that test scores would rise and solution times fall for subjects using data–oriented exception handling. Data analysis shows that subjects achieved higher average test scores in shorter time for version D programs than for version C programs. Dividing the questions into two groups (comprehension and modification), we determined that the differences of total scores mainly come from the latter group. Thus, our data–oriented exception handling mechanism may have greater impact on modification activities (a more realistic programming task) than on a programmer's ability to understand and answer questions about a program.

4.2. Summary

Declaring exceptions with a type's operations and associating handlers with objects in declarations centralize information about exceptional processing and separate it from algorithmic processing. Experimental results indicate that data–oriented exception handling can be used to produce programs that are smaller, better structured, and easier to understand and modify. With the exception of pre–processing time, no significant time or space penalty results from this change. We are currently working on proof rules and test coverage metrics that will permit us to compare these alternative exception handling mechanisms in other ways.

5. Compiler Scheduling of Parallel Computation

Much of the existing work in parallelism detection assumes scientific code with DO loop indices, shared memory or restricted single–assignment, dataflow languages. We are extending existing detection work to conventional languages (e.g., C) that may execute on either shared or non–shared memory architectures [Bennet90].

5.1. Control State Analysis

A *control state* for a program is a set of control state variable–value pairs that fully determine its control decisions. Our model of a computation is a digraph where each node is a unique control state consuming and producing values, and each arc represents the data passed between the nodes. Costs are associated with each arc (interprocessor communication), and node (execution time). Before a task starts it must take time to receive from all parent tasks which are on different processors, and after it completes it must take time to send to all its children which are on different processors.

Our prototype parallelizing compiler consists of a parser, allocator, compressor, and code generator. The parser reads source programs annotated with function timing information and unrolls loops to produce a digraph representation called a "static graph" in which nodes represent control states and arcs represent data dependencies. This graph is sent to an allocator which chooses an allocation of graph nodes to processors which is a tradeoff between the speedup due to parallel execution and the overhead losses due to the communication required for parallel execution. The control trees chosen by the allocator are sent to a compressor, which merges identical trees and examines branches for repetitive structures that can be turned into loops. Finally, the code generator which produces object code from the compressed trees, inserting necessary communication primitives. The resulting program is a truly distributed implementation of the original algorithm. Nodes communicate independently rather than under the synchrony of a master processor.

5.2. Empirical Studies

We are currently experimenting with examples to evaluate how well our system works. Preliminary evidence shows that we achieve speedups comparable to those achieved by DO–loop analysis when working on problems whose control states contain just DO–loop indices and the program counter. For example, analyzing a simple numerical integration using the trapezoidal rule, our system achieved a factor of 4.7 improvement on 7–processor 68000–based machine. Our model predicts slightly better improvement may be realized when when more processors are available to us for testing. Furthermore, we are able to improve while statement programs that other systems cannot handle. (e.g., a program that uses a sum to approximate an integral which is especially useful near the integrand's spikes).

Several research problems related to granularity and serialization of operations have arisen. When the granularity of the operations in loop bodies is small compared to communications operations, communications costs quickly destroy any gains from parallelism. We are currently investigating techniques for allocating basic blocks rather than individual statements by either compressing adjacent statements or treating inner loops atomically.

When a loop's values are functions of the results from its previous iteration, parallel execution may prove no better than serial execution. Only sophisticated analyses of the functions (producing new algorithms) could improve such programs. In contrast, some serialization problems (e.g., those in a Taylor series program we examined), stem only from the need to tell that an iteration exists before it can be executed. Since our control–state method is based on

preserving the set of control states created by an equivalent serial computation, these programs apparently cause problems. However, a run–ahead mechanism that would permit control states to be created and discarded if they were unneeded may improve the performance of such programs.

6. AFOSR Sponsored Publications

(1) [Antoy90] Antoy, S., P. Forcheri, M. Molfino, and M. Zelkowitz, Rapid prototyping of system enhancements, 1st IEEE Int. Conf. on System Integration, (April 1990), 7 pp. *(to appear).*

(2) [Basili89] Basili, V.R., Software Development: A Paradigm for the Future, *Proc. of the 13th Annual Comp. Soc. Int. Computer Software and Appl. Conf.,* Orlando FL, September, (1989), 471–484 (Keynote Address).

(3) [Bennet90] Bennet, T.W., Using Control States for Parallelism Extraction. University of Maryland, Department of Computer Science, TR–2385, Ph.D. thesis, (January 1990).

(4) [Cardenas90] Cardenas, S. and M.V. Zelkowitz, Evaluation criteria for functional specifications, ACM/IEEE 12th Int. Conf. on Software Eng., Nice Fr (March, 1990), 8 pp. *(to appear).*

(5) [Cui89] Cui, Q., Data–Oriented Exception Handling. University of Maryland, Department of Computer Science, TR–2384, Ph.D. Thesis, (November 1989).

(6) [Cui90] Cui, Q. and J.D. Gannon. Data–Oriented Exception Handling in Ada, IEEE Int. Conf. on Computer Languages, New Orleans LA (March 1990), (to appear).

(7) [Mills89] Mills, H.D., V.R. Basili, J.D. Gannon, R.G. Hamlet. A first course in computer science: mathematical principles for software engineering. *IEEE Trans. Soft. Eng.* (May 1989), 550–559.

(8) [Zelkowitz89a] Zelkowitz, M V., L. Herman, D. Itkin, B. Kowalchack, A tool for understanding program execution, *J. of Pascal, Ada and Modula-2* 8,3 (1989) 10–20.

(9) [Zelkowitz89b] Zelkowitz, M., L. Herman, D. Itkin, and B. Kowalchack, Experiences building a syntax–directed editor *Software Engineering J.,* 4, 6 (November, 1989) 294–300.

(10) [Zelkowitz90a] Zelkowitz, M.V. Evolution towards a specifications environment: Experiences with syntax editors, *Information and Software Technology,* (April, 1990) *(to appear).*

(11) [Zelkowitz90b] Zelkowitz, M.V., A functional correctness approach towards program development, *(Submitted for publication).*

This Appendix represents activities during the period March 1, 1987 through June 30, 1988. It had previously been reported to AFOSR.

TECHNICAL SUMMARY

AFOSR GRANT 87-0130
March 1, 1987 – June 30, 1988

Principal Investigators:
Victor R. Basili
John D. Gannon
Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Date: November 8, 1988

Abstract

This report summarizes the activites from this grant during the period March 1, 1987 through June 30, 1988. Research progressed on the following topics: (1) Development of the TAME measurement environment, a tool being developed to exploit the G-Q-M model for effective management control over large scale software development; (2) Preliminary results from using the SUPPORT Pascal programming environment and what was learned from its use over the last three years; (3) Initial ideas in extending the integrated environment approach towards a specifications methodology; (4) Understanding the semantics of programs maintained in an integrated environment; (5) Further investigations in verification methodology based upon functional correctness properties and (6) Properties of distributed systems.

This report represents the technical summary of AFOSR grant 87-0130 to the Department of Computer Science of the University of Maryland for the period March 1, 1987 until June 30, 1988. Copies of all relevant papers have previously been forwarded to AFOSR as they appeared. Since it is a large multi-investigator effort, some of these summaries represent the completion of previous projects while others represent ideas just now being developed.

This research was under the direction of the principal investigators Dr. Victor Basili, Dr. John Gannon and Dr. Marvin Zelkowitz. In addition, Dr. Gregory Johnson received support from this grant and directed some of its activities.

1. The TAME Environment

TAME (Tailoring A Measurement Environment) is a project to automate as much as possible the measurement, feedback and planning needed in the context of software development [Basili 87a], [Basili 87c]. The development of a prototype TAME system is currently underway. The TAME system provides a mechanism for managers and engineers to develop project-specific goals, and generate operational definitions based upon these goals that specify the appropriate metrics needed for evaluation. The evaluation and feedback can be done in real time as well as help prepare for post mortems. It will help in the tailoring of the software development process [Basili 87b].

The TAME requirements have been developed and a model architecture has been proposed [Basili 87c], [Basili 88]. TAME consists of four major components: user interface, an evaluation mechanism, a measurement mechanism and an information base. The user interface provides the support needed for helping managers develop operational goals (i.e., the tools necessary to operationally define goals based upon the goal/question/metric paradigm, to input and validate manually collected data, and to physically interface with the system). The evaluation mechanism provides feedback based upon the various project goals. The measurement mechanism provides capabilities for automated collection of metrics (e.g., structural test coverage metrics [Wu 87], data binding metrics, and static source code metrics from Ada programs). The information base contains the historical database, all project documents, the current project database, a goal/question/metric database and other information necessary for support.

We are developing a first TAME prototype addressing the specific measurement, feedback and planning needs. We are building upon previous experience in measuring and evaluating aspects of the software development process and product. We will be using the system for various measurement projects to evaluate the system during its various prototyping phases.

The scope of the TAME system is ambitious. On the one hand, we can build upon experience in measurement and evaluation. On the other hand, a large number of research questions remain to be solved. One major project is the further development of the Goal/Question/Metric Paradigm so that goals can be generated more easily and reused more effectively. To aid in this process, we have chosen the development of a set of goals, questions and metrics based upon evaluating a requirements document in the form of data flow diagrams. The idea is to provide an example where we can develop a specific set of product and process goals and generalize toward the generation of a goal generation language that supports the generation of goals and their interpretation. Work has begun, with Professor Amiram Yehudai from Tel Aviv University in Israel. We are using his automated object oriented model to encode the requirements process so that it can be characterized and

evaluated. We are analyzing both process conformance, based on the model characterization and product qualities, based upon an encoding of the final product.

The goal generation language will be implemented using some tool such as a hypertext system, such as HyperCard, or a knowledge based system, such as Kee.

It is assumed that the first TAME prototype will evolve over time and that we will learn a great deal from formalizing the various aspects of the TAME project as well as integrating the various paradigms, subsystems, or individual tools that will necessarily be built.

2. Environments

Research activities in the integrated environment area has progressed on two fronts, leading to one new initiative. The SUPPORT integrated environment activities matured, leading to new work in the area of specification systems and reuse. In addition, research continued in the areas of formal semantics for integrated environments.

2.1. Integrated Environments

The SUPPORT integrated environment project is nearing the end of its initial phase of activities. SUPPORT is an integrated environment for the development of Pascal programs that is operational on both Berkeley UNIX and IBM PC environments. In order to gain experience in using such an environment, based upon syntax-directed editing technology, it has been used for the past 3 years as the primary environment by beginning Computer Science majors at the University of Maryland. Based upon this experience, numerous changes and enhancements have been made to this system.

An appropriate user interface is crucial for the acceptance of any environmental tool [Zelkowitz 88a]. SUPPORT provides a seamless interface to about a dozen tools that enable the user to build, modify, test, debug and document his program. The consistent command structure and consistent screen display is necessary for ease in learning and accepting such systems. SUPPORT is built around a multiwindowed display process, and the value of this design decision has proven itself numerous times as it has proven to be easy to add new windows and to give the user easy flexibility of altering the viewing screen.

SUPPORT implements its own window manager, thus transporting the code to other machine architectures has proven to be a relatively easy process. If the project were restarted today, then a machine independent window system, such as the MIT consortium's X-windows, would probably be used; however, such a system did not exist when SUPPORT development began. SUPPORT has clearly shown the value of such a machine independent screen interface and such standardization efforts should continue.

The ability to use any system implies that the user must be able to think in terms of the language being implemented. This includes diagnostic facilities, also. Because of this, a diagnostic tool Drs. has been implemented as part of the SUPPORT environment [Kowalchack 87]. With Drs. the user can nagivate through a Pascal activation record in terms of the source program data structures. The model implemented is the usual one — each procedure creates an array of data objects and these data objects are nested as each procedure calls other procedures. Drs. allows the programmer to interact with the SUPPORT run-time system without resorting to primitive machine architecture concepts and makes debugging programs much easier. The ability to "reverse engineer" the Pascal source data objects from the actual activation record storage is essentially instantaneous, and while there is a performance penalty for

providing such a feature, in a diagnostic setting, machine performance is of relatively low priority. No detailed performance statistics have been generated; however, running the system on an IBM PC/AT, the trace window of machine execution executes about as fast as a user can monitor it, and the Drs. window operates at the speed of the keyboard, which is needed for the user to indicate nagigation commands.

An additional feature that has proven to be valuable is the invisibility of the syntax editor. One criticism of such systems is that users have to input source text essentially as a program parse tree. However, this is unnatural since most programmers think in terms of linear strings of source statements. In SUPPORT, however, a user can either input the program in the *traditional* syntax–editing paradigm as a parse tree, or else input the normal linear source text. An internal LALR parser converts such source text to the parse tree representation automatically. Thus many of the undesirable aspects of syntax editors is hidden from the user.

In addition, experience with the development of SUPPORT has validated many important guidelines for large scale system development [Zelkowitz 88c]. While SUPPORT is a relatively small system (about 28,000 lines of Pascal source program in 28 modules), aspects of this should apply to larger developments. Some of these guidelines are:

(1) *Use interpreted tables for configuration control.* In SUPPORT the grammar that is processed as well as the format of the screen display are data files interpreted during program execution. This allows for dynamic updating of the files and much easier changing of the format than if these algorithms were built in. The higher execution time of such tables in general are not apparent.

(2) *Data abstractions simplify designs.* SUPPORT separates each major data item in a separate module. Screen management became almost a trivial operation as the functions to add and delete lines from a window were processed independently from the mechanism that displayed windows on the screen. The former allowed an easy interface to adding new windows while the latter allowed for easy transportability of the system to alternative machine architectures. A second such feature was that the underlying program tree was independent of what was displayed on the screen, allowing for each to change independently.

(3) *Use simple data structures.* Efficiency is determined by overall design, not by clever coding.

(4) *Windows are powerful display mechanisms.* This has already been discussed.

(5) *Understand your environment.* This seemingly simple statement has major implications. The literature generally proposes a top down design methodology for software development; however, few such systems are truly 100% top down. As part of the specifications, there are some constraints about the underlying hardware and software that will exist. For example, if you have a task of traveling from Washington to San Francisco, then a top down solution would probably not even consider a horse–drawn stagecoach since such vehicles no longer exist. Similarly, any computer solution must necessarily *ab initio* have certain constraints based upon the eventual low level design. In [Zelkowitz 88b] and [Zelkowitz 88c] several examples are given of how the architecture of the IBM PC and the Intel 8088 microprocessor dictated several high level design constraints — something that probably would not have occurred in a purely top down design methodology.

2.2. Executable Specifications

Results from using SUPPORT have tended to confirm that experienced users do not view the entering of source text as a major productivity enhancer. In studying this problem, it appears as if such a system, like SUPPORT, would be more effective as a specifications and design tool. Towards this end, the AS* project has been started [Antoy 87]. AS* is a system for converting formal specifications into executable programs. Such a system has two major benefits: (1) The specifications are formal, and can be proven to have certain properties; and (2) The executable specifications serve as a check (as a prototyping tool or as a testing tool) against with the eventual source program can be compared. This provides a mechanism for "viewing" a system long before the source code is ready.

Our model is based upon algebraic specifications. A data type (i.e. a *sort*) is defined with a series of *operations* giving the transformations on the data and a series of *constructors* for building objects of the type. Operations are defined by a series of algebraic *axioms*. By considering each axiom set as a series of rewrite rules, they can be converted into executable statements in a language like Pascal.

For example, the following is a formal definition of a list:

```
sort list is
constructor
      nil;
      cons : integer, list;
operation head : list -> integer is axiom
   head(nil) == ?;
   head(cons(X,Y)) == X;
   end;
```

A list is constructed out of the empty list (e.g., *nil*) and the *cons* constructor. A single operation *head* for the head of a list is demonstrated here.

A system ASSUPPORT, nominally based upon SUPPORT, has been designed to investigate this technology. A SUPPORT grammar for creating sorts was designed, a Prolog program, ASVERIFY, was proposed for automatically validating the sort axioms for termination and other verification properties, and ASPC was designed to translate the sorts into standard Pascal. Development of these tools has been initiated and will continue in the future.

2.3. Semantic Models

The approach of denotational semantics has begun to have a visible effect on the design of programming languages. The designs of languages as diverse as ML, Scheme, and CLU have been influenced by the mathematical insights of the denotational community. Mills's functional semantics and the programming methodology based on it also reflect the denotational approach [Mills 87]. Over the past year we have explored the possibilities of creating programming environments that reflect the denotational approach. We have implemented a prototype environment and language called GL to serve as a testbed for the research [Johnson 87] [Johnson 88].

The goal of the research is to create an entire integrated programming environment all of whose aspects reflect the underlying simplicity and elegance of a careful, mathematically rigorous design based on denotational semantics. The motivation for this effort is the hypothesis that such an environment will have a significant positive impact on programmer

productivity. While the Mills approach primarily focuses on the specification and implementation phases of the software life–cycle, the GL effort is directed toward the test, debug, and maintenance phases. The research has been aimed at the following two goals:

(1) to assess on a relatively small but still realistic scale the impact on programmer productivity of the new proposed directions in programming environments, and

(2) to demonstrate the feasibility of the approach (i.e., to show that the proposed powerful mechanisms that the environment will make available to the user can be implemented with reasonable efficiency).

The GL environment is built around the idea that both stores and continuations should be obtainable and manipulable from inside the programming environment. GL provides a controlled execution environment for programs. As with standard program testing facilities such as dbx, a program can be read into the environment and executed in steps or under the control of breakpoints. Unlike conventional program test facilities, when a program is at a pause in its execution it is possible to obtain from the environment the extant continuation and the current store. The continuation is a function representing the remainder of the computation, and so in a sense a continuation represents the computation's future. A continuation is in effect a snapshot of the control state extant when a program has been stopped. Similarly, the contents of the store constitute a snapshot of the data state of the partially executed program. It is quite convenient, when an executing program is stopped, to dynamically capture the full continuation extant at that point and experimentally apply it to different stores. Similarly, it is often convenient to apply the partial continuation representing completion of the most recently invoked function or the most recent few functions to a variety of different stores. The power of the environment to allow the user to save and experimentally manipulate both the control state and data state of a program under study provide a new and interesting paradigm of reasoning about programs.

2.4. A programming environment based on types–as–propositions

Constructive type theory is being actively investigated by several groups as a new basis for program development. This approach represents what might be called the ultimate in strongly typed programming languages. The type system of such a language is much more powerful than that of a conventional language, and in fact it is sufficiently powerful that it can be used as a specification language. Hence the term 'types as propositions:' A type becomes a logical proposition asserting that a program provides a certain function. To specify a desired program, one simply gives the type of the program. The type system is sufficiently general and powerful that virtually all computational tasks can be thus specified. The problem of program verification then reduces to type checking: If a given program can be type checked and shown to be of the appropriate type, this constitutes a proof that the program satisfies the specification to which the type corresponds. This research is beginning to have an impact on the design of programming languages. For instance the module structure of Standard ML, with its product and existential types, reflects this influence. In this approach data types, logical expressions, and algorithms are all treated uniformly in a single elegant framework. In such a system, Coquand's calculus of constructions for instance, logical expressions are lambda terms of a certain sort, programs are lambda terms, and data structures are represented using lambda terms. So type correctness and program proofs simply involve checking relationhips of various sorts among lambda terms. This conceptual unity gives the approach much of its power and appeal.

We are curently looking at possible implications for programming environments of the new approach. Our past research in the area of programming environments and that of others has indicated that one of the problems of current programming environments is that they are too inflexible. An environment that supports a given programming paradigm often becomes a straight–jacket, imposing that paradigm even in places where it is awkward or inappropriate. We are in the early stages of investigating a new multi–paradigm environment based on types as propositions. Instead of enforcing a single, fixed paradigm, the environment and its language will support a spectrum of styles. At one extreme is a PRL–like top down approach for creating a constructive proof of given proposition and hence an algorithm that meets the specification corresponding to the proof. Alternatively it will be possible to build the program directly, in a more Hoare–like style in which the algorithm and its proof are developed in tandem. Finally, it will be possible to provide trusted algorithms without proof. The programmer should have the flexibility to be able to shift paradigms, eliciting from the support environment the sort of assistance that is most appropriate at any given point in the development process. Moreover, there should be a natural, seamless transition between paradigms. We hypothesize that such an environment will enable programmers to achieve a new level of productivity; different programming tasks require different sorts of automated support, and in a system such as the one we are investigating the programmer will be able to obtain the sorts or assistance that will most enhance his productivity in a very dynamic, task–driven way [Duggan 88].

3. Programming Methodology

Program verification technology has generally centered upon Hoare–like predicate calculus axioms or algebraic specifications. Research has progressed on using a functional methodology as an alternative to these techniques. The methodology depends upon a simple execution trace table as the mechanism to verify a proof, so it is relatively easy to build tools that use this technique. Research so far has centered upon development of the basic proof methodology [Mills 87] [Gannon 87] [Mills 89].

The basic theory is developed as follows. Let p be a program. Let f be a function. Then we say that f is the function executed by program p if $\boxed{p} = f$. $\boxed{p}$ is defined to be the transformation that maps a given program state vector (e.g., all of the variables in p and their associated values) into another state vector. If program p is the sequence: *a;b;c;d* ..., then $\boxed{p} = \boxed{a} \circ \boxed{b} \circ \boxed{c} \circ \boxed{d} \circ \ldots$

IF and assignment statements are straightforward (e.g., see [Gannon 87]). The recursive nature of the WHILE adds some complexity to the process. The basic conditions to insure that a function f is equivalent to the WHILE statement:

```
WHILE b DO
      d
```

are:

(A.1) $f = \boxed{\text{if b then d}} \circ f$.

(A.2) domain (f) = domain($\boxed{\text{while b do d}}$).

(A.3) (Not(b) → f) = (Not(b) → *Identity*).

Given the above three conditions, we can define a mechanism to *develop* a WHILE loop that meets a given functional specification f:

(B.1) range(f) $\subset$ domain(f).

(B.2) if x $\in$ range(f) then $f(x) = x$.

(B.3) Find a b such that $\boxed{b}$ evaluates to true in domain(f) – range(f).

(B.4) $\boxed{b}$ evaluates to false in range(f).

(B.5) Develop d so all values needed for f are preserved in d.

(B.6) Show that the loop must terminate. Hence $\boxed{\text{while b do d}}$ is defined on an appropriate domain.

Research on refining these rules is continuing [Zelkowitz 88d].

4. A Scheduling Compiler

Much of the existing work in detecting parallelism assumes SIMD hardware or global memories. Many existing MIMD machines of significant size are constructed of processors which must communicate by message–passing or have global memories with some long access times.

We analyze programs written in conventional languages for single processors and transform them to implementations for MIMD machines that minimize execution times [Bennet 88]. The analysis is performed on the control states of a computation. These states contain receive inputs and produce outputs (including the next computation state).

The steps of the analysis are:

(1) Unroll WHILE statements a finite number of times to obtain a static graph representing all executions. Each node represents a control state on a separate processor with a cost that describes the time it executes assuming its input data are already present on its processor. Before starting, a node must take time to receive from its parents on different processors, and after it completes it must take time to send results to its children.

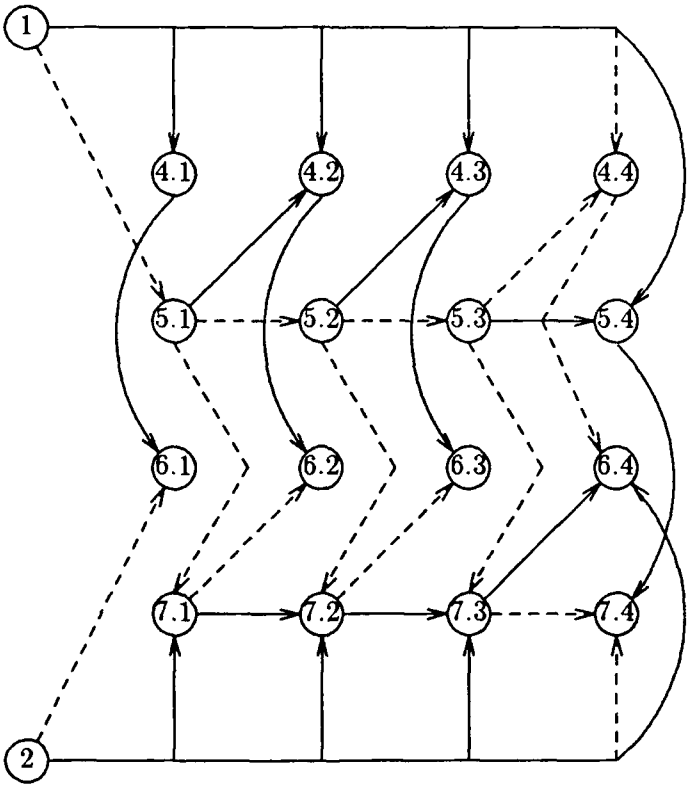
(2) Using continuation probabilities and a heuristic search, draw control trace arcs between control states to be executed by a single processor. Assigning different states to the same processor cuts communication costs, but also reduces the amount of parallelism.

(3) For each control trace, generate code inserting communication primitives as needed. If there are fewer processors than control traces, some control traces will have to be combined before code is generated.

Consider the following example conversion program.

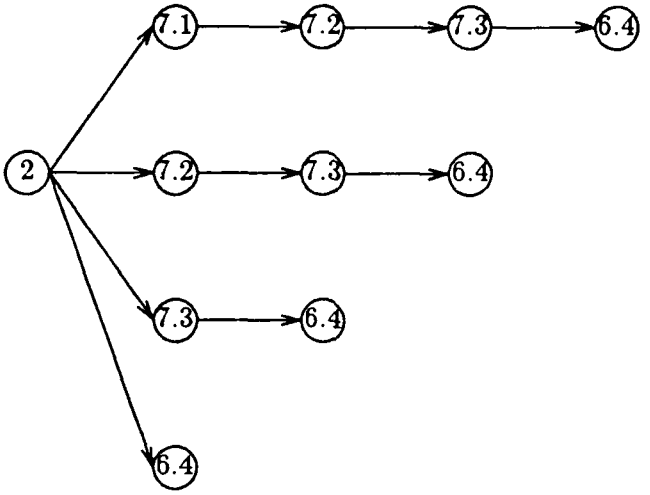
```
1     num = M;
2     sub = N;
3     while num != 0 do {
4        dig = num mod 10;
5        num = num div 10;
6        arr[sub] = chr(ord('0') + dig);
7        sub = sub - 1
8     }
```

In the figure below, nodes labelled 4.1, 4.2, 4.3, and 4.4 represent the last, next to last, second to last, and all previous executions of statement 4. Solid lines represent control traces that should be assigned to a single processor. Nodes connected by dashed lines are each run on separate processors.

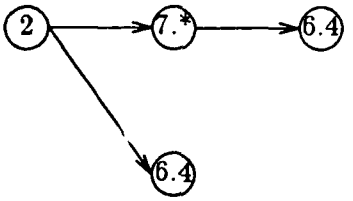


Most of the control traces are obvious definition–use pairings except that statement 4 must transmit the value of **num** to statement 6 so that the latter can decide if it needs to send the value of **sub** to one of its successors.

Choosing the control trace starting at statement 2, we obtain the following subgraph.



which can be compressed to:



Finally code is generated from the compressed subgraph.

```
      receive(num);
2     sub = N;
      if num div 10 != 0 {
         do {
             receive(num);
7            sub = sub - 1;
             send(T34,sub);}
         while num != 0;
         receive(dig);
6        arr[sub] = chr(ord('0') + dig);
         send(T5, arr[sub])
         }
      else {
         receive(dig);
6        arr[sub] = chr(ord('0') + dig);
         send(T1, arr[sub])
         }
```

The code corresponding to statement 7.i waits to receive num from a node containing statement 5.i and sends the value of sub to a receive statement inserted before the code for statement 6 in a node executing the sequence of instructions <5.i, 4.i+1, 6.i+1>.

We are continuing work toward building a prototype compiler. We are also trying to improve our heuristic for handling conditional statements within loops. While these statements cause other extraction mechanisms severe problems, they are easily represented within our formalism. However, we need a much better heuristic search algorithm to analyze the graphs.

5. AFOSR–Supported Publications

(1) [Antoy 87] S. Antoy, P. Forcheri, B. Kowalchack, M. Molfino, S. Pearlman, M. Zelkowitz, Using abstractions in a Pascal environment, AICA Conference, Trento, Italy (September, 1987) 155–162.

(2) [Basili 87a] V. R. Basili and H. D. Rombach, TAME: Tailoring and Ada Measurement Environment. *Proc. of the 5th National Conference on Ada Technology,* Arlington, VA (March 1987) 318–325.

(3) [Basili 87b] V.R. Basili and H.D. Rombach, Tailoring the Software Process to Project Goals and Environments. *Proc. of the 9th Int. Conf. on Software Engineering,* Monterey, CA (April 1987) 345–357.

(4) [Basili 87c] V.R. Basili and H.D. Rombach, TAME: Integrating Measurement into SW Environments. University of Maryland, Department of Computer Science, TR 1764 (July 1987).

(5) [Basili 88] V. R. Basili, H. D. Rombach, The TAME Project: Towards Improvement–Oriented Software Environments, *IEEE Trans. on Software Engineering* 14, 6, (June 1988) 758–773.

(6) [Bennet 88] T.W. Bennet. Using Control States for Paralellism Extraction. *Proc. of the 1988 Parallel Processing Conference* 2, (August, 1988), 135–139.

(7) [Duggan 88] Duggan, D., A programming environment based on types–as–specifications, University of Maryland Technical report UMIACS–TR–88–69 (September 1988).

(8) [Gannon 88] Gannon J. D., R. G. Hamlet, H. D. Mills, Theory of Modules, *IEEE Trans. on Software Engineering* 13, 7 (1987) 820–829.

(9) [Johnson 87] Johnson G. F., GL – A Denotational Testbed with Continuations and Partial Continuations as First–class Objects *Proc. ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques,* (June, 1987) 165–176.

(10) [Johnson 88] Johnson G. F., D. Duggan, Stores and Partial Continuations as First–Class Objects in a Language and its Environment *Proc. ACM Symposium on Principles of Programming Languages,* (January 1988).

(11) [Kowalchack 87] Kowalchack B. and M. V. Zelkowitz, Drs. A language–oriented diagnostic run–time system, *The Role of Language in Problem Solving 2,* Elsevier Science Publishers (1987) 377–390.

(12) [Mills 87] Mills H., V. Basili, J. Gannon, R. Hamlet, *Principles of Computer Programming, A Mathematical Approach,* William Brown (1987).

(13) [Mills 89] Mills H. D., V. R. Basili, J. D. Gannon, R. G. Hamlet, Mathematical principles for a first course in software engineering, *IEEE Trans. on Software Engineering,* (1989) *(to appear).*

(14) [Wu 87] L. Wu, V. R. Basili, and K. Reed, A Structure Coverage Tool for Ada Software Systems. *Proc. of the 5th National Conference on Ada Technology,* Arlington, VA (March 1987) 294–303.

(15) [Zelkowitz 88a] M. V. Zelkowitz, L. Herman, D. Itkin, B. Kowalchack, A tool for understanding program execution *J. of Pascal, Ada and Modula-2* (1988) (to appear).

(16) [Zelkowitz 88b] Zelkowitz M. V., et al, A SUPPORT tool for teaching computer programming, in *Educational Issues in Software Engineering,* Springer verlag (1988) *(to appear).*

(17) [Zelkowitz 88c] M. V. Zelkowitz, L. Herman, D. Itkin, B. Kowalchack, Guidelines for software development *(submitted for publication).*

(18) [Zelkowitz 88d] Zelkowitz M. V., A functional correctness approach towards program development, *(in preparation).*